CS103L SPRING 2020

UNIT 4: FUNCTIONS

TEXT

FUNCTIONS

- Functions are encapsulated pieces of code - mini programs
- Also called procedures or methods
- Perform a computation given some inputs, usually return a value (result)
  - When using functions we treat as black-box
  - We care *what* they do, not *how*
    - Ex: double a = sin(x);
    - There are many ways to compute or calculate sin(x), as long as we get the right answer back
  - This is actually a useful, powerful concept
  - Functions can be re-written, optimized, improved, without changing the code that *uses* the function

TEXT

ATTRIBUTES OF A FUNCTION

- Has a name to identify the function: avg, sin, max, min
- Zero or more inputs
- Zero or one output
  - Note, only *one* output
- Performs a computation - code is in between { }
- Statements execute sequentially - like all C++ code
- Function is defined once, can be called as many times as necessary
- One function can call another, can call another, and so on

TEXT

EXECUTING A FUNCTION

- When a function is called, calling code is “paused” while function is executed
- Function executes based on the inputs given
- When the function returns, the expression containing the function call evaluates to the return value.

THIS EXPRESSION EVALUATES TO 6

THIS EXPRESSION EVALUATES TO 199

- When a function hits a return statement, it immediately stops (returns) with the given value.
- Non-void functions must have at least one return statement that sets the return value
- Void functions may have zero or more return statements (no value allowed)

```
#include <iostream>
using namespace std;

int max(int a, int b)
{
  if(a > b)
    return a;
  else
    return b;
}


int main(int argc, char *argv[])
{
   int x=6, z;
   z = max(x,4);
   cout << "Max is " << z << endl;
   z = max(125, 199);
   cout << "Max is " << z << endl;
   return 0;
}
```

FUNCTION CALLS

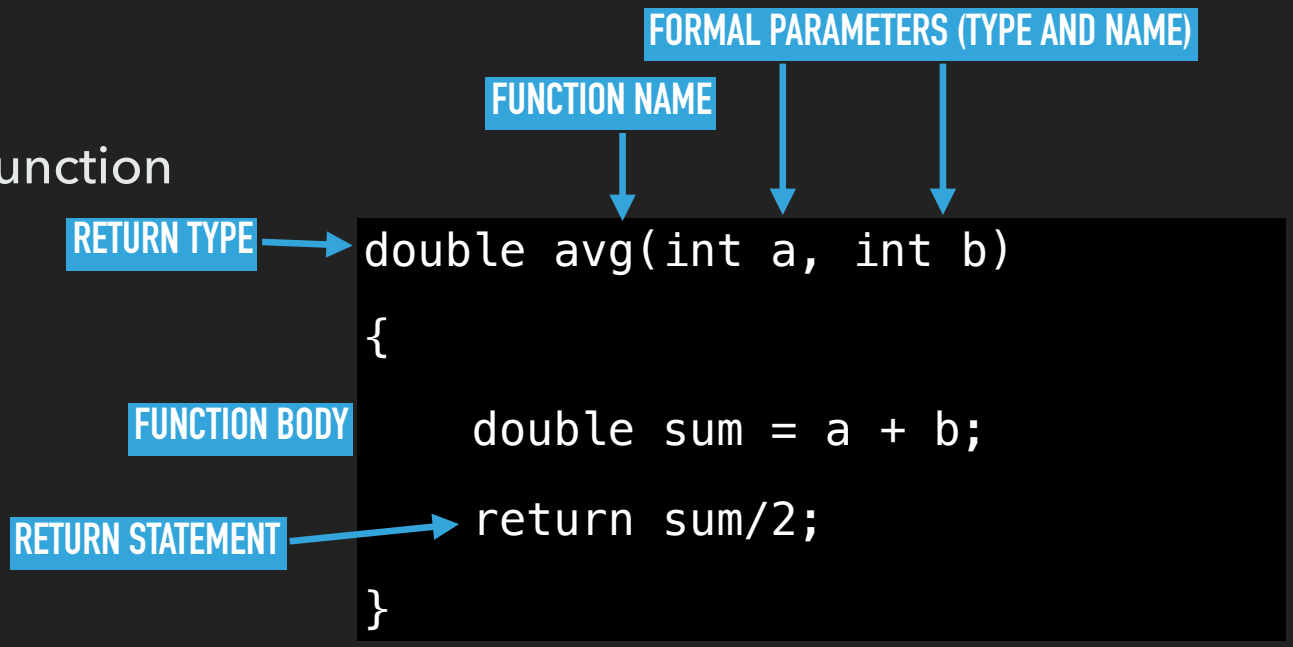
- Function calls can be used like any expression
- Ex: min(a,b) || max(c,d)
- Ex: 1.0 + sin(x)/cos(y);
- Ex: max of three numbers?

```
int x=5,y=10,z=20;
//max of x,y,z?
int m = max(max(x,y),z);
```

TEXT

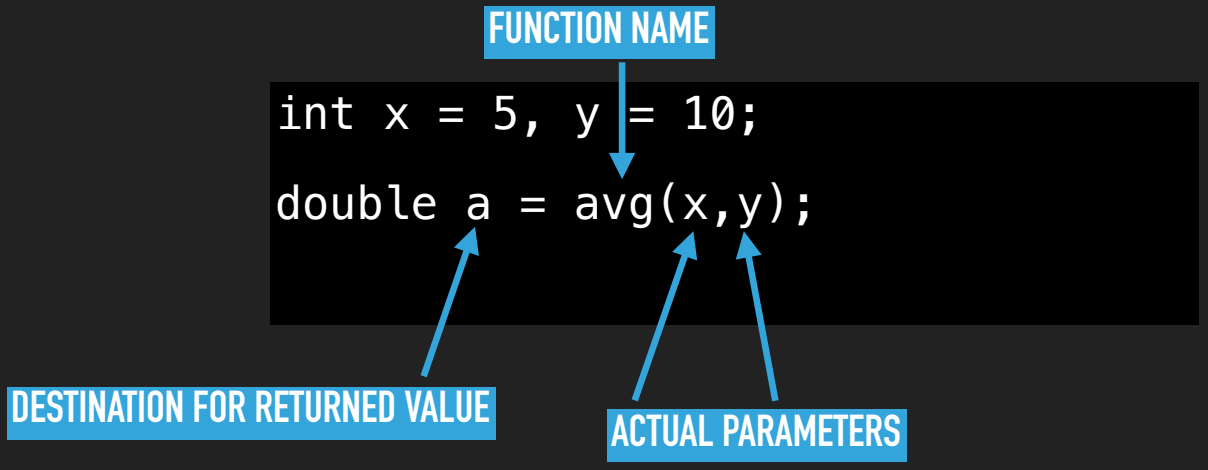
ANATOMY OF A FUNCTION DEFINITION

- Formal parameters are the inputs when you define the function
- Have a type and name
- Are local variables to the function



ANATOMY OF A FUNCTION CALL

- Actual parameters are the values of what is passed to the function when it is called
- Important to note: A *copy* of the actual parameter is given to the function



PASS BY VALUE

- Functions in C/C++ defined this way are pass-by-value
- A *copy* of the actual parameter is given to the function
- Nothing happens to the actual parameter in the caller
- What does this code do?
- How many x's do we have?
- Are they the same?

```
#include <iostream>
using namespace std;

void inc(int x)
{
    x = x+1;
}

int main()
{
    int x = 6;
    inc(x);
    cout << x << endl;
}
```

TEXT

PROGRAM DECOMPOSITION

- C is a procedural language. Procedures are the basic unit of abstraction: programs are broken down into a set of procedures, called in some order to solve a problem.
- Functions (procedures, methods) are units of code that can be called from other pieces of code, taking inputs and producing outputs.
- C++ is considered “object oriented” - but we can still use functions
  - We’ll get to the difference later in the semester

TEXT

EXERCISE – DECOMPOSITION TECHNIQUES

- When developing your recipe, plan or algorithm
- List out the **verbs** and/or **tasks** that make up the solution to the problem
- Ex: modeling (simulating) a Blackjack casino game?
  - shuffle(), deal(), bet(), double_down()…
- Ex: a program that models social networks?
  - addUser(), addFriend(), updateStatus()…

FUNCTION DEFINITIONS AND COMPILERS

- C/C++ compilers are single-pass, top-to-bottom
- The compiler needs to “know” about something before it can be used
- What happens here?

```
int main()
{
    double area;
    area = triangle_area(5.0, 3.5);
}

double triangle_area(double b, double h)
{
    return 0.5*b*h;
}
```

TEXT

FUNCTION DEFINITIONS SOLUTION #1

- Move function definitions above main
- Not considered the best solution.
- Why?

```
double triangle_area(double b, double h)
{
    return 0.5*b*h;
}

int main()
{
    double area;
    area = triangle_area(5.0, 3.5);
}
```

FUNCTION PROTOTYPES

- Better solution:
  - prototype (declare) function before main
  - Implement anywhere
- Why is this better?
- Prototypes are like a promise to the compiler: “Hey, compiler, I’m eventually going to define this and it will look like this…”
- After seeing the prototype the compiler can compile code that uses the function before it even sees the implementation

```
double triangle_area(double, double);

int main()
{
    double area;
    area = triangle_area(5.0, 3.5);
}

double triangle_area(double b, double h)
{
    return 0.5*b*h;
}
```

TEXT

NEED FOR FUNCTION PROTOTYPES

- Get in the habit of using prototypes, it will save you frustration and is good programming practice
- Consider the following two functions
- Called “mutually recursive” - 104/170 topic
- Can’t be done without prototypes

```
funcA()
{
    if( condition )
        funcB();
    return;
}

funcB()
{
    if( condition )
        funcA();
    return;
}
```

TEXT

FUNCTION SIGNATURES

- A signature is can uniquely identify you
- Functions have a signature:
  - name
  - number and type of arguments
- Two functions can have the same name! (as long as they have different signatures over all)
  - int f1(int), int f1(double), int f1(int, double), int f1(int, char), double f1(), void f1(char)
  - All of these specify different functions called f1 - don’t do this ;-p

TEXT

FUNCTION OVERLOADS

- Two functions with the same name, but different signatures are said to be “overloaded”
- Which is easier?

- int max_int(int, int)
- double max_double(double, double)
- int pow_ints(int, int)
- double pow(double, double)

OVERLOADED VERSIONS

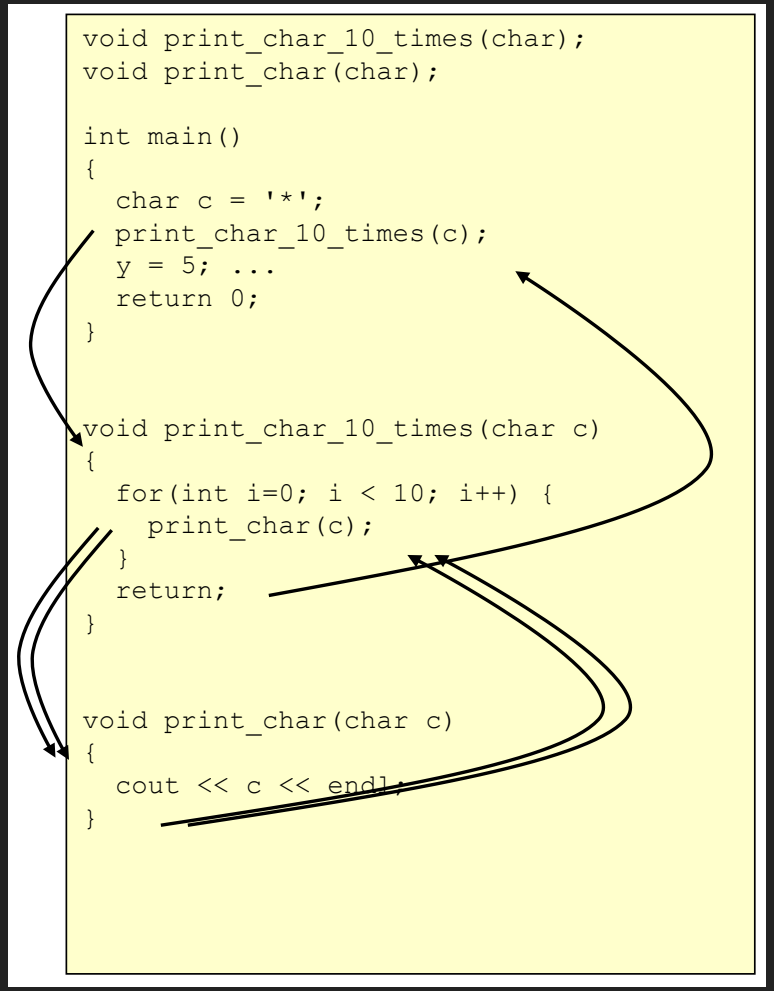
- int max(int, int)
- double max(double, double)
- int pow(int, int)
- double pow(double, double)

IN CLASS EXERCISES

- abs_func
- Remove Factor
- ASCII square
- overloading

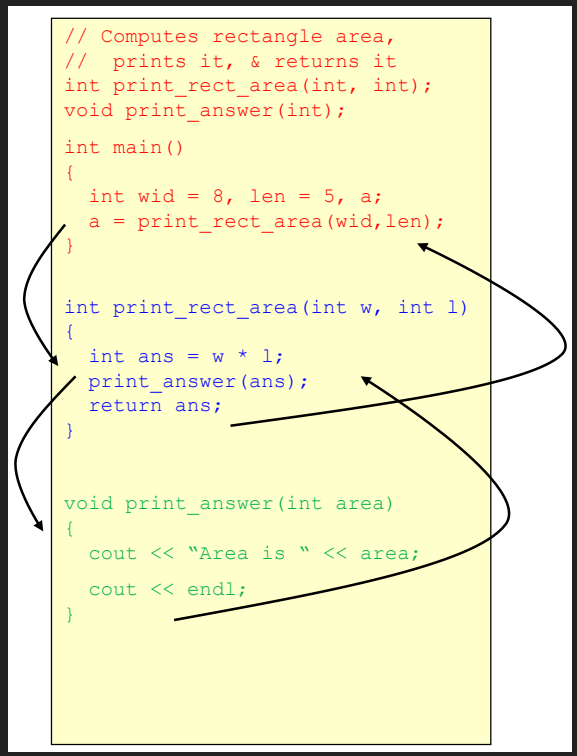
FUNCTION CALL SEQUENCING

- Functions can call other functions
- Each calling function is “paused” while the called function is executed
- When the function finishes the calling function resumes
- Each function call has it’s own “scope”, variables inside the function are only visible/ accessible to that invocation



ANOTHER SEQUENCING EXAMPLES

- Since one function can call another, and that can call another how does the compiler keep everything straight?

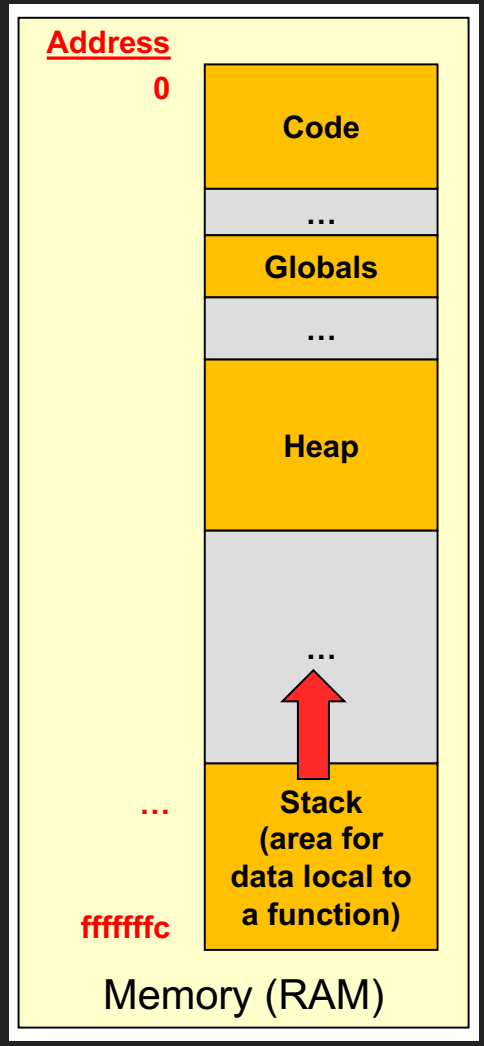


```
// Computes rectangle area,
//  prints it, & returns it
int print_rect_area(int, int);
void print_answer(int);

int main()
{
  int wid = 8, len = 5, a;
  a = print_rect_area(wid,len);
}


int print_rect_area(int w, int l)
{
  int ans = w * l;
  print_answer(ans);
  return ans;
}


void print_answer(int area)
{
  cout << "Area is " << area;

  cout << endl;
}
```

```
funcA(int x)
{
   if( x>0 )
      funcB(x-1);
   return;
}

funcB(int x)
{
   if( x>0 )
      funcA(x-1);
   return;
}

int main()
{
    int x=2;
    funcB(x);
}
```

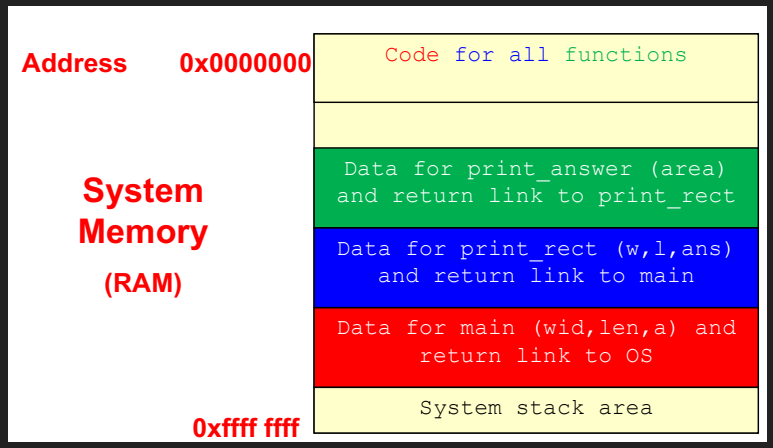
COMPUTER MEMORY ORGANIZATION

- To answer that we need to see how memory is organized for your program
- Entire memory address space (here 32-bits) is broken up and assigned for different purposes
- Compiled code goes at the bottom (near address 0)
- Then global variables are assigned some space
- Then the heap (discussed later in the semester)
- Then the stack



THE STACK

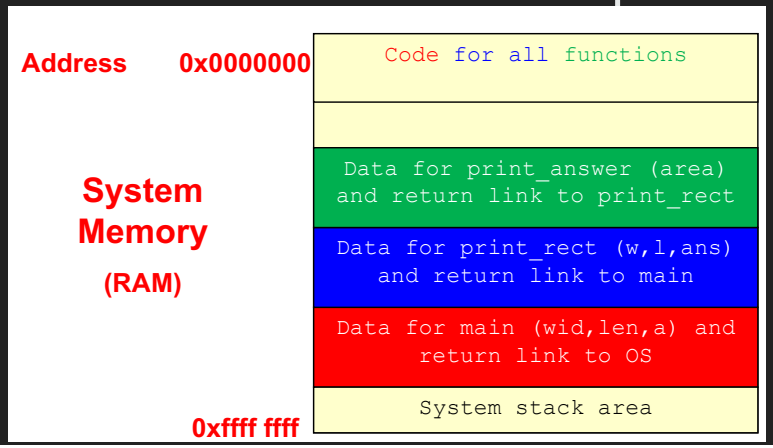
- The stack is what we care about here
- The stack is segmented into pieces to hold the data (variables) for each function. Why they are called “stack-local” variables
- Each time a function is called, a new “stack frame” is allocated. The code running for that function only has access to variables in it’s stack frame
- When one function is finished the stack frame is deallocated and control returns to the function below it on the stack




```
// Computes rectangle area,
//  prints it, & returns it
int print_rect_area(int, int);
void print_answer(int);

int main()
{
  int wid = 8, len = 5, a;
  a = print_rect_area(wid,len);
}


int print_rect_area(int w, int l)
{
  int ans = w * l;
  print_answer(ans);
  return ans;
}


void print_answer(int area)
{
  cout << "Area is " << area;

  cout << endl;
}
```

LOCAL VARIABLES AND SCOPE

- Variables defined in a function are “local” to that function
- They are said to only be “in scope” inside the function
- These variables “live” in the stack frame for the function
- “Die” or go out of scope when the function completes



TEXT

SCOPE

- All variables in C/C++ have a scope
  - Context in which they are a valid identifier
  - Global variables are valid anywhere in your .cpp file
  - Variables defined inside a { } block are valid inside that block
  - Including:
    - { } of a function
    - { } of an if statement
    - { } of a loop

SCOPING EXAMPLE

- These scoping rules mean you can have variables with the same name, valid in the same scope
- If this is the case, the closest (inner most) scope is used
- How many x's are in this code?
- When where are they valid?

```
#include <iostream>
using namespace std;

int x = 5;

int main()
{
  int a, x = 8, y = 3;
  cout << "x = " << x << endl;
  for(int i=0; i < 10; i++){
    int j = 1;
    j = 2*i + 1;
    a += j;
  }
  a = doit(y);
  cout << "a=" << a ;
  cout << "y=" << y << endl;
  cout << "glob. x" << ::x << endl;
}


int doit(int x)
{
   x--;
   return x;
}
```

TEXT

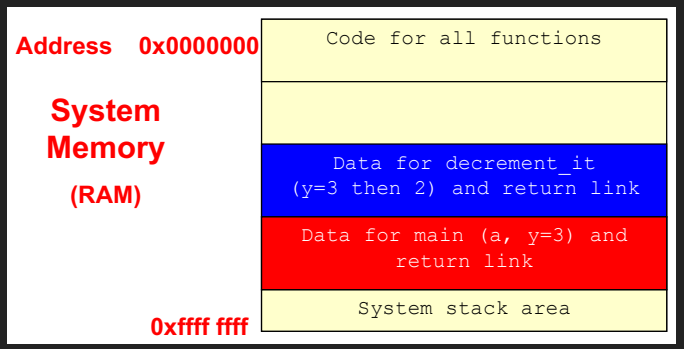
PASS BY VALUE

- Earlier we mentioned functions in C/C++ are pass by value
- Passing a value to an argument of a function makes a copy
  - like e-mailing a document, any changes made by recipient won't reflect in your local copy
  - they have to e-mail back (return) the document

PASS BY VALUE AND THE STACK

- Now we can see why function calls are pass by value
- The actual parameters live in calling function
- Copies are placed into the formal parameters, which are in the stack frame for the function
- Operations on the formal parameters local to that stack frame

```
void decrement_it(int);

int main()
{
  int a, y = 3;
  decrement_it(y);
  cout << "y = " << y << endl;
  return 0;
}

void decrement_it(int y)
{
   y--;
}
```



IN CLASS EXERCISES

- vowels